Saros: Reducing Dependency Duplication in Decentralized Serialization-Graph Validation

Leon Tsuboi[1,a)] Jun Nemoto[2,b)] Hideyuki Kawashima[1,c)]

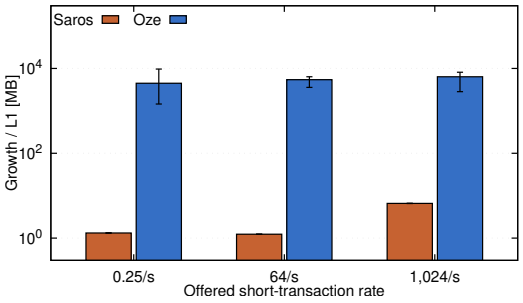
Abstract: For long update transactions, Oze decentralizes multi-version serialization graph (MVSG) validation across records, but its validators copy dependencies into every record they visit. Saros stores each dependency fact once and lets records inherit it through history-prefix references. Preliminary results on BoMB show that Saros grows memory 960–4,400× less per committed long transaction and validates long transactions 370–1,330× faster than Oze.

1. Introduction

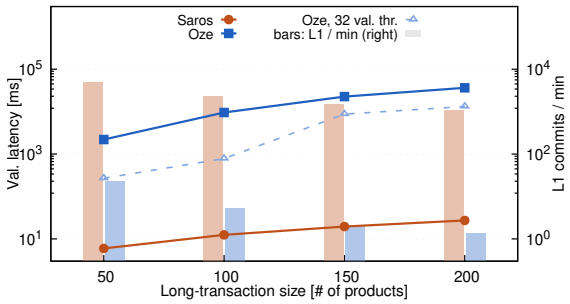
Product costing in manufacturing runs long update transactions alongside short ones [1]: a costing transaction walks each product's bill of materials (BoM) and reads thousands of material costs, while short transactions keep updating them. Graph databases face the same tension [2]. Optimistic and multi-version protocols [3], [4], [5], [6], [7] repeatedly abort such long transactions: on the BoM benchmark (BoMB) [1], their abort rate reaches 27–100% in our runs. Locking holds every lock until commit, so short transactions that update what the long one read must wait for it, and deterministic locking [8] also needs read and write sets in advance, which a BoM walk cannot know once the BoM changes.

Multi-version serialization graph (MVSG) validation [9] avoids these problems. Transactions need no read or write sets in advance and hold no locks between operations: a read latches its record only while reading, and writes are buffered until commit. At commit, a validator checks that their dependencies still admit a serial order. Oze [1] decentralizes the graph across records, so that validators at different records do not contend.

However, a cycle can pass through dependencies held at several records, so validation cannot stay local: Oze's validator merges its accumulated dependencies into the graph of every record it visits and leaves the result for later validators. We call the resulting copies *dependency duplication*. As BoMB's long transaction grows from 25 to 100 products, each read-dependency fact ends up in 1,756–5,613 record graphs on average. Oze's memory grows by 4.5–6.3 GB per committed long transaction (Fig. 1a), exhausting a 64 GiB budget within 1.5–10 minutes, and validation, at least 99.7% of the long transaction's latency, grows superlinearly from 2 s at 50 products to 36 s at 200 (Fig. 1b). Parallel validation spreads the copying over threads but does not reduce it.



(a) Memory growth per committed long transaction



(b) Validation latency (lines) and L1 throughput (bars)

Fig. 1 Costs of dependency duplication in Oze and their removal by Saros on BoMB (preliminary results). (a) Memory growth per committed long transaction (L1) at three short-transaction loads (600-second runs, three per point; whiskers: min–max). (b) Median validation latency (lines, left axis) and L1 commits per minute (bars, right axis) as L1 grows (90-second runs). Oze with 32 validation threads exhausts 64 GiB at 50 and 100 products; only its latency is shown.

We present *Saros*, a dependency representation for decentralized MVSG validation that stores each fact once and refers to it elsewhere. Saros changes only how its *host* protocol (Oze) stores and recovers inherited dependencies, and keeps the host's validation rules.

2. Saros Design

Histories and prefix references. Each record r keeps an

[1] Keio University, Fujisawa, Japan
[2] Scalar, Inc., Tokyo, Japan
[a)] leon.tsuboi@keio.jp
[b)] jun.nemoto@scalar-labs.com
[c)] river@sfc.keio.ac.jp

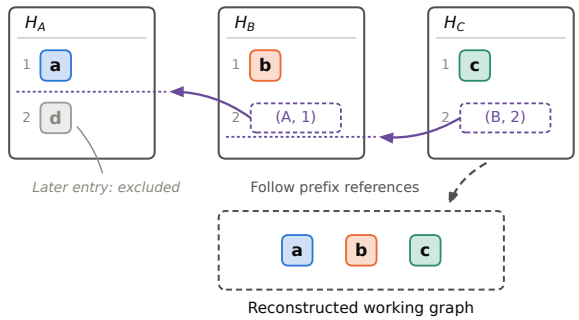


Fig. 2 Instead of copying earlier facts into each later record as Oze does, Saros appends prefix references $(A, 1)$ and $(B, 2)$, which a later validator follows to reconstruct a working graph. Fact d, appended after the cited cutoff, is excluded.

append-only history H_r of facts about r: transaction identities, read facts, dependency edges, and abort markers. Entries receive increasing sequence numbers and never change once published. A *prefix reference* (r, k) stands for the first k entries of H_r; its cutoff k is fixed at creation, so later appends do not change what it contributes. Where Oze merges a validator's accumulated graph into each record it visits, Saros appends one reference to the record where the validator last published (Fig. 2). The references form a chain from which a later validator can rebuild everything the earlier one gathered.

Validation. At each record it visits, while holding the record's lock, a validator *reconstructs* the record's evidence into a private working graph by importing the published prefix and, recursively, the prefixes its references name; checks the graph for a cycle; appends a reference to its previous publication; and discovers further records to visit by Oze's rule. Because read facts stay at the records that were read, discovery takes reachable transactions' read sets from a *footprint registry* that stores each read set once. Reading a referenced prefix needs no lock, because published entries are immutable.

Transitive retirement. An old reference can lead to newer evidence, so retiring it by its creator's age would hide that evidence. Each entry d has a retirement bound $\rho(d)$, the epoch of the transaction it describes (for an edge, the later endpoint's), and a reference created by transaction T receives

$$\rho(\mathrm{ref}(r,k)) = \max\Big(\mathrm{epoch}(T),\ \max_{1\le i\le k} \rho(H_r[i])\Big),$$

tracked as a running prefix maximum. A watermark advances over leading entries whose bounds lie below the host's epoch floor, and imports skip them.

Correctness. Saros applies the host's procedures to a working graph that, at each record, holds at least the evidence Oze's record graph would hold, so it never commits a cycle that Oze would detect.

3. Preliminary Evaluation

We implemented Saros in Oze's code base and ran BoMB on 40 hardware threads of an Intel Xeon Gold 5418N socket. BoMB's long transaction, L1, costs one factory's products (100 by default, about 7,200 reads); its score is the highest short-transaction throughput at which L1 aborts in under 1% of attempts. Oze and Saros use identical validation rules.

Saros grows memory by 1.2–6.6 MB per committed L1, against 4.5–6.3 GB for Oze (Fig. 1a), and stays within the 64 GiB budget over 600-second runs, although its memory still grows because retired history is not yet physically reclaimed. As L1 grows from 50 to 200 products, Saros validates it in 6–27 ms, 370–1,330× faster than Oze and 45–480× faster than Oze with 32 validation threads, and commits 1,070–4,871 L1 per minute against Oze's 1–22 (Fig. 1b). Against Silo, TicToc, MOCC, ERMIA, Cicada, and deterministic locking, Saros commits 1.4–2.7× more long transactions than the best of them on a static BoM and 3–5× more on a dynamic BoM, where short transactions also change the BoM. On a static BoM, deterministic locking, given every key in advance, sustains higher short-transaction throughput from eight threads up. On a dynamic BoM, however, its L1 must read the BoM to predict its keys and aborts whenever the BoM changes before it locks them: up to 99% of L1 attempts abort at a few requests per second, and its score falls to at most 112 transactions per minute, against 1.7k–97k for Saros.

4. Conclusion

Preliminary results indicate that storing each dependency fact once removes most of the memory and validation time that dependency duplication costs. Next, we plan to compare Saros with TuskFlow [2] on graph workloads.

Acknowledgments This work was supported by the New Energy and Industrial Technology Development Organization (NEDO) under the Research and Development Project of the Enhanced Infrastructures for Post-5G Information and Communication Systems (JPNP20017) and JPNP25014; JSPS KAKENHI Grant Number 25H00446; JST CREST Grant Number JPMJCR24R4; the SECOM Science and Technology Foundation; JST COI-NEXT SQAI (JPMJPF2221); and JST Moonshot R&D Grant Number JPMJMS2215.

References

[1] Nemoto, J., Kambayashi, T., Hoshino, T. and Kawashima, H.: Oze: Decentralized Graph-Based Concurrency Control for Long-Running Update Transactions, *PVLDB*, Vol. 18, No. 8, pp. 2321–2333 (2025).
[2] Theodorakis, G., Firth, H., Clarkson, J., Crooks, N. and Webber, J.: TuskFlow: An Efficient Graph Database for Long-Running Transactions, *PVLDB*, Vol. 18, No. 12, pp. 4777–4790 (2025).
[3] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy Transactions in Multicore In-Memory Databases, *Proc. SOSP*, pp. 18–32 (2013).
[4] Yu, X., Pavlo, A., Sanchez, D. and Devadas, S.: TicToc: Time Traveling Optimistic Concurrency Control, *Proc. SIGMOD*, pp. 1629–1642 (2016).
[5] Wang, T. and Kimura, H.: Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores, *PVLDB*, Vol. 10, No. 2, pp. 49–60 (2016).
[6] Kim, K., Wang, T., Johnson, R. and Pandis, I.: ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads, *Proc. SIGMOD*, pp. 1675–1687 (2016).
[7] Lim, H., Kaminsky, M. and Andersen, D. G.: Cicada: Dependably Fast Multi-Core In-Memory Transactions, *Proc. SIGMOD*, pp. 21–35 (2017).
[8] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D. J.: Calvin: Fast Distributed Transactions for Partitioned Database Systems, *Proc. SIGMOD*, pp. 1–12 (2012).
[9] Bernstein, P. A., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).